EXHIBIT M

blob: 7cd4fba9c9abe57f7b85c7ed788dba2b9eb963af [file] [log] [blame]

```
/*
 * Copyright (C) 2012 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.android.tools.lint.checks;

import com.android.annotations.NonNull;
import com.android.annotations.Nullable;
import com.android.tools.lint.checks.ControlFlowGraph.Node;
import com.android.tools.lint.detector.api.Category;
import com.android.tools.lint.detector.api.ClassContext;
import com.android.tools.lint.detector.api.ClassScanner;
import com.android.tools.lint.detector.api.Context;
import com.android.tools.lint.detector.api.Detector;
import com.android.tools.lint.detector.api.Implementation;
import com.android.tools.lint.detector.api.Issue;
import com.android.tools.lint.detector.api.JavaContext;
import com.android.tools.lint.detector.api.Lint;
import com.android.tools.lint.detector.api.LintFix;
import com.android.tools.lint.detector.api.Location;
import com.android.tools.lint.detector.api.Scope;
import com.android.tools.lint.detector.api.Severity;
import com.android.tools.lint.detector.api.SourceCodeScanner;
import com.intellij.psi.PsiClass;
import com.intellij.psi.PsiMethod;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import org.jetbrains.uast.UCallExpression;
import org.objectweb.asm.Opcodes;
import org.objectweb.asm.tree.AbstractInsnNode;
import org.objectweb.asm.tree.ClassNode;
import org.objectweb.asm.tree.InsnList;
import org.objectweb.asm.tree.JumpInsnNode;
import org.objectweb.asm.tree.LdcInsnNode;
import org.objectweb.asm.tree.MethodInsnNode;
import org.objectweb.asm.tree.MethodNode;
import org.objectweb.asm.tree.analysis.AnalyzerException;

/**
 * Checks for problems with wakelocks (such as failing to release them) which can lead to
 * unnecessary battery usage.
 */
public class WakelockDetector extends Detector implements ClassScanner, SourceCodeScanner {
    public static final String ANDROID_APP_ACTIVITY = "android.app.Activity";

    /** Problems using wakelocks */
    public static final Issue ISSUE =
            Issue.create(
                            "Wakelock",
                            "Incorrect `WakeLock` usage",
                            "Failing to release a wakelock properly can keep the Android device in "
                                    + "a high power mode, which reduces battery life. There are several causes "
                                    + "of this, such as releasing the wake lock in `onDestroy()` instead of in "
                                    + "`onPause()`, failing to call `release()` in all possible code paths after "
                                    + "an `acquire()`, and so on.\n"
                                    + "\n"
                                    + "NOTE: If you are using the lock just to keep the screen on, you should "
                                    + "strongly consider using `FLAG_KEEP_SCREEN_ON` instead. This window flag "
                                    + "will be correctly managed by the platform as the user moves between "
                                    + "applications and doesn't require a special permission. See "
                                    + "https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html#FLAG_KEEP_SCREEN_ON.",
                            Category.PERFORMANCE,
                            9,
                            Severity.WARNING,
                            new Implementation(WakelockDetector.class, Scope.CLASS_FILE_SCOPE))
                    .setAndroidSpecific(true);

    /** Using non-timeout version of wakelock acquire */
    public static final Issue TIMEOUT =
            Issue.create(
                            "WakelockTimeout",
```

```
                        "Using wakeLock without timeout",
                        "Wakelocks have two acquire methods: one with a timeout, and one without. "
                                + "You should generally always use the one with a timeout. A typical "

                                + "timeout is 10 minutes. If the task takes longer than it is critical "
                                + "that it happens (i.e. can't use `JobScheduler`) then maybe they "
                                + "should consider a foreground service instead (which is a stronger "
                                + "run guarantee and lets the user know something long/important is "
                                + "happening).",
                        Category.PERFORMANCE,
                        9,
                        Severity.WARNING,
                        new Implementation(WakelockDetector.class, Scope.JAVA_FILE_SCOPE))
                .setAndroidSpecific(true);

    private static final String WAKELOCK_OWNER = "android/os/PowerManager$WakeLock";
    private static final String RELEASE_METHOD = "release";
    private static final String ACQUIRE_METHOD = "acquire";
    private static final String IS_HELD_METHOD = "isHeld";
    private static final String POWER_MANAGER = "android/os/PowerManager";
    private static final String NEW_WAKE_LOCK_METHOD = "newWakeLock";

    /** Constructs a new {@link WakelockDetector} */
    public WakelockDetector() {}

    @Override
    public void afterCheckRootProject(@NonNull Context context) {
        if (mHasAcquire && !mHasRelease && context.getDriver().getPhase() == 1) {
            // Gather positions of the acquire calls
            context.getDriver().requestRepeat(this, Scope.CLASS_FILE_SCOPE);
        }
    }

    // ---- Implements ClassScanner ----

    /** Whether any {@code acquire()} calls have been encountered */
    private boolean mHasAcquire;

    /** Whether any {@code release()} calls have been encountered */
    private boolean mHasRelease;

    @Override
    @Nullable
    public List<String> getApplicableCallNames() {
        return Arrays.asList(ACQUIRE_METHOD, RELEASE_METHOD, NEW_WAKE_LOCK_METHOD);
    }

    @Override
    public void checkCall(
            @NonNull ClassContext context,
            @NonNull ClassNode classNode,
            @NonNull MethodNode method,
            @NonNull MethodInsnNode call) {
        if (!context.getProject().getReportIssues()) {
            // If this is a library project not being analyzed, ignore it
            return;
        }

        if (call.owner.equals(WAKELOCK_OWNER)) {
            String name = call.name;
            if (name.equals(ACQUIRE_METHOD)) {
                if (call.desc.equals(
                        "(J)V")) { // acquire(long timeout) does not require a corresponding release
                    return;
                }
                mHasAcquire = true;

                if (context.getDriver().getPhase() == 2) {
                    assert !mHasRelease;
                    context.report(
                            ISSUE,
                            method,
                            call,
                            context.getLocation(call),
                            "Found a wakelock `acquire()` but no `release()` calls anywhere");
                } else {
                    assert context.getDriver().getPhase() == 1;
                    // Perform flow analysis in this method to see if we're
                    // performing an acquire/release block, where there are code paths
                    // between the acquire and release which can result in the
                    // release call not getting reached.
                    checkFlow(context, classNode, method, call);
                }
            } else if (name.equals(RELEASE_METHOD)) {
                mHasRelease = true;

                // See if the release is happening in an onDestroy method, in an activity.
                if ("onDestroy".equals(method.name)) {
                    PsiClass psiClass = context.findPsiClass(classNode);
                    PsiClass activityClass = context.findPsiClass(ANDROID_APP_ACTIVITY);
                    if (psiClass != null
                            && activityClass != null
                            && psiClass.isInheritor(activityClass, true)) {
```

```
                    context.report(
                            ISSUE,
                            method,
                            call,
                            context.getLocation(call),
                            "Wakelocks should be released in `onPause`, not `onDestroy`");
                }
            }
        }
    } else if (call.owner.equals(POWER_MANAGER)) {
        if (call.name.equals(NEW_WAKE_LOCK_METHOD)) {
            AbstractInsnNode prev = Lint.getPrevInstruction(call);
            if (prev == null) {
                return;
            }
            prev = Lint.getPrevInstruction(prev);
            if (prev == null || prev.getOpcode() != Opcodes.LDC) {
                return;
            }
            LdcInsnNode ldc = (LdcInsnNode) prev;
            Object constant = ldc.cst;
            if (constant instanceof Integer) {
                int flag = (Integer) constant;
                // Constant values are copied into the bytecode so we have to compare
                // values; however, that means the values are part of the API
                final int PARTIAL_WAKE_LOCK = 0x00000001;
                final int ACQUIRE_CAUSES_WAKEUP = 0x10000000;
                final int both = PARTIAL_WAKE_LOCK | ACQUIRE_CAUSES_WAKEUP;
                if ((flag & both) == both) {
                    context.report(
                            ISSUE,
                            method,
                            call,
                            context.getLocation(call),
                            "Should not set both `PARTIAL_WAKE_LOCK` and `ACQUIRE_CAUSES_WAKEUP`. "
                                    + "If you do not want the screen to turn on, get rid of "
                                    + "`ACQUIRE_CAUSES_WAKEUP`");
                }
            }
        }
    }
}

private static void checkFlow(
        @NonNull ClassContext context,
        @NonNull ClassNode classNode,
        @NonNull MethodNode method,
        @NonNull MethodInsnNode acquire) {
    final InsnList instructions = method.instructions;
    MethodInsnNode release = null;

    // Find release call
    for (int i = 0, n = instructions.size(); i < n; i++) {
        AbstractInsnNode instruction = instructions.get(i);
        int type = instruction.getType();
        if (type == AbstractInsnNode.METHOD_INSN) {
            MethodInsnNode call = (MethodInsnNode) instruction;
            if (call.name.equals(RELEASE_METHOD) && call.owner.equals(WAKELOCK_OWNER)) {
                release = call;
                break;
            }
        }
    }

    if (release == null) {
        // Didn't find both acquire and release in this method; no point in doing
        // local flow analysis
        return;
    }

    try {
        MyGraph graph = new MyGraph();
        ControlFlowGraph.create(graph, classNode, method);

        int status = dfs(graph.getNode(acquire));
        if ((status & SEEN_RETURN) != 0) {
            String message;
            if ((status & SEEN_EXCEPTION) != 0) {
                message = "The `release()` call is not always reached (via exceptional flow)";
            } else {
                message = "The `release()` call is not always reached";
            }

            context.report(ISSUE, method, acquire, context.getLocation(release), message);
        }
    } catch (AnalyzerException e) {
        context.log(e, null);
    }
}

private static final int SEEN_TARGET = 1;
private static final int SEEN_BRANCH = 2;
private static final int SEEN_EXCEPTION = 4;
```

```
270     private static final int SEEN_RETURN = 8;
271
272     /** TODO RENAME */
273     private static class MyGraph extends ControlFlowGraph {
274         @Override
275         protected void add(@NonNull AbstractInsnNode from, @NonNull AbstractInsnNode to) {
276             if (from.getOpcode() == Opcodes.IFNULL) {
277                 JumpInsnNode jump = (JumpInsnNode) from;
278                 if (jump.label == to) {
279                     // Skip jump targets on null if it's surrounding the release call
280                     //
281                     //  if (lock != null) {
282                     //      lock.release();
283                     //  }
284                     //
285                     // The above shouldn't be considered a scenario where release() may not
286                     // be called.
287                     AbstractInsnNode next = Lint.getNextInstruction(from);
288                     if (next != null && next.getType() == AbstractInsnNode.VAR_INSN) {
289                         next = Lint.getNextInstruction(next);
290                         if (next != null && next.getType() == AbstractInsnNode.METHOD_INSN) {
291                             MethodInsnNode method = (MethodInsnNode) next;
292                             if (method.name.equals(RELEASE_METHOD)
293                                     && method.owner.equals(WAKELOCK_OWNER)) {
294                                 // This isn't entirely correct; this will also trigger
295                                 // for "if (lock == null) { lock.release(); }" but that's
296                                 // not likely (and caught by other null checking in tools)
297                                 return;
298                             }
299                         }
300                     }
301                 }
302             } else if (from.getOpcode() == Opcodes.IFEQ) {
303                 JumpInsnNode jump = (JumpInsnNode) from;
304                 if (jump.label == to) {
305                     AbstractInsnNode prev = Lint.getPrevInstruction(from);
306                     if (prev != null && prev.getType() == AbstractInsnNode.METHOD_INSN) {
307                         MethodInsnNode method = (MethodInsnNode) prev;
308                         if (method.name.equals(IS_HELD_METHOD)
309                                 && method.owner.equals(WAKELOCK_OWNER)) {
310                             AbstractInsnNode next = Lint.getNextInstruction(from);
311                             if (next != null) {
312                                 super.add(from, next);
313                                 return;
314                             }
315                         }
316                     }
317                 }
318             }
319
320             super.add(from, to);
321         }
322     }
323
324     /**
325      * Search from the given node towards the target; return false if we reach an exit point such as
326      * a return or a call on the way there that is not within a try/catch clause.
327      *
328      * @param node the current node
329      * @return true if the target was reached XXX RETURN VALUES ARE WRONG AS OF RIGHT NOW
330      */
331     protected static int dfs(ControlFlowGraph.Node node) {
332         AbstractInsnNode instruction = node.instruction;
333         if (instruction.getType() == AbstractInsnNode.JUMP_INSN) {
334             int opcode = instruction.getOpcode();
335             if (opcode == Opcodes.RETURN
336                     || opcode == Opcodes.ARETURN
337                     || opcode == Opcodes.LRETURN
338                     || opcode == Opcodes.IRETURN
339                     || opcode == Opcodes.DRETURN
340                     || opcode == Opcodes.FRETURN
341                     || opcode == Opcodes.ATHROW) {
342                 return SEEN_RETURN;
343             }
344         }
345
346         // There are no cycles, so no *NEED* for this, though it does avoid
347         // researching shared labels. However, it makes debugging harder (no re-entry)
348         // so this is only done when debugging is off
349         if (node.visit != 0) {
350             return 0;
351         }
352         node.visit = 1;
353
354         // Look for the target. This is any method call node which is a release on the
355         // lock (later also check it's the same instance, though that's harder).
356         // This is because finally blocks tend to be inlined so from a single try/catch/finally
357         // with a release() in the finally, the bytecode can contain multiple repeated
358         // (inlined) release() calls.
359         if (instruction.getType() == AbstractInsnNode.METHOD_INSN) {
360             MethodInsnNode method = (MethodInsnNode) instruction;
361             if (method.name.equals(RELEASE_METHOD) && method.owner.equals(WAKELOCK_OWNER)) {
362                 return SEEN_TARGET;
```

```
            } else if (method.name.equals(ACQUIRE_METHOD) && method.owner.equals(WAKELOCK_OWNER)) {
                // OK
            } else if (method.name.equals(IS_HELD_METHOD) && method.owner.equals(WAKELOCK_OWNER)) {
                // OK
            } else {
                // Some non acquire/release method call: if this is not associated with a
                // try-catch block, it would mean the exception would exit the method,
                // which would be an error
                if (node.exceptions.isEmpty()) {
                    // Look up the corresponding frame, if any
                    AbstractInsnNode curr = method.getPrevious();
                    boolean foundFrame = false;
                    while (curr != null) {
                        if (curr.getType() == AbstractInsnNode.FRAME) {
                            foundFrame = true;
                            break;
                        }
                        curr = curr.getPrevious();
                    }

                    if (!foundFrame) {
                        return SEEN_RETURN;
                    }
                }
            }
        }

        // if (node.instruction is a call, and the call is not caught by
        // a try/catch block (provided the release is not inside the try/catch block)
        // then return false
        int status = 0;

        boolean implicitReturn = true;
        List<Node> successors = node.successors;
        List<Node> exceptions = node.exceptions;
        if (!exceptions.isEmpty()) {
            implicitReturn = false;
        }
        for (Node successor : exceptions) {
            status = dfs(successor) | status;
            if ((status & SEEN_RETURN) != 0) {
                return status;
            }
        }

        if (status != 0) {
            status |= SEEN_EXCEPTION;
        }

        if (!successors.isEmpty()) {
            implicitReturn = false;
            if (successors.size() > 1) {
                status |= SEEN_BRANCH;
            }
        }
        for (Node successor : successors) {
            status = dfs(successor) | status;
            if ((status & SEEN_RETURN) != 0) {
                return status;
            }
        }

        if (implicitReturn) {
            status |= SEEN_RETURN;
        }

        return status;
    }

    // Check for the non-timeout version of wakelock acquire

    @Nullable
    @Override
    public List<String> getApplicableMethodNames() {
        return Collections.singletonList("acquire");
    }

    @Override
    public void visitMethodCall(
            @NonNull JavaContext context,
            @NonNull UCallExpression call,
            @NonNull PsiMethod method) {
        if (call.getValueArgumentCount() > 0) {
            return;
        }

        if (!context.getEvaluator().isMemberInClass(method, "android.os.PowerManager.WakeLock")) {
            return;
        }

        Location location = context.getLocation(call);
        LintFix fix =
                fix().name("Set timeout to 10 minutes")
```

```
456                     .replace()
457                     .pattern("acquire\\s*\\(()\\s*\\)")
458                     .with("10*60*1000L /*10 minutes*/")
459                     .build();
460
461         context.report(
462                 TIMEOUT,
463                 call,
464                 location,
465                 ""
466                         + "Provide a timeout when requesting a wakelock with "
467                         + "`PowerManager.Wakelock.acquire(long timeout)`. This will ensure the OS will "
468                         + "cleanup any wakelocks that last longer than you intend, and will save your "
469                         + "user's battery.",
470                 fix);
471     }
472 }
```